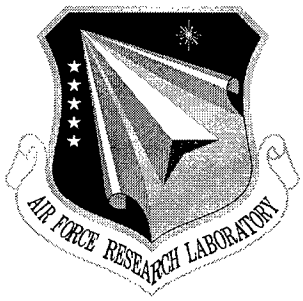
AFRL-IF-RS-TR-1998-70
Final Technical Report
May 1998



A THEORETICAL FRAMEWORK FOR SPECIFICATION REFINEMENT VIA TRANSFORMATIONS

Florida International University

Paul Attie

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-1998-70 has been reviewed and is approved for publication.

APPROVED:

MARK GERKEN, Capt, USAF
Project Engineer

FOR THE DIRECTOR:

NORTHRUP FOWLER, III, Technical Advisor
Information Technology Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTD, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 1998	Final Jul 96 - Aug 97

4. TITLE AND SUBTITLE

A THEORETICAL FRAMEWORK FOR SPECIFICATION REFINEMENT VIA TRANSFORMATIONS

5. FUNDING NUMBERS

C - F30602-96-1-0249
PE - 61102F
PR - 2304
TA - FR
WU - P2

6. AUTHOR(S)

Paul Attie

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Florida International University
School of Computer Science
Miami FL 33199

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/IFTD
525 Brooks Road
Rome, NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-1998-70

11. SUPPLEMENTARY NOTES

Air Force Research Laboratory Project Engineer: Mark Gerken, Capt, USAF/IFTD/(315) 330-2974

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The idea of successively refining an abstract specification until it contains enough detail to suggest an implementation has been investigated by numerous researchers. The emphasis to date has been on techniques that, unfortunately, lead to a large amount of manual labor for each refinement step. With such techniques, both the cost and the possibility of errors arising in the formal manipulation are high. Using a theorem prover can reduce the number of manipulation errors, but the amount of labor involved is daunting. This research explores an alternative solution to the refinement problem, namely the use of syntactic transformations to realize each refinement step. We reduce formal labor by employing automatic transformations that guarantee the preservation of desirable properties such as deadlock freedom, safety, and liveness of process-algebraic specifications. This report presents three syntactic transformations that can be used to replace an atomic action in a concurrent program by a program fragment. We include applicability conditions for these transformations, and show that deadlock freedom and certain liveness properties are preserved when the transformations are applied in a context where the applicability conditions are satisfied.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Concurrent Program, Deadlock-Freedom, Design Transformation, Multiparty Interaction, Refinement, Formal Methods	36
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

Abstract

The idea of successively refining an abstract specification until it contains enough details to suggest an implementation has been investigated by numerous researchers. The emphasis to date has been on techniques that, unfortunately, lead to a large amount of manual formal labor for each refinement step. With such techniques, both the cost and the possibility of errors arising in formal manipulation are high. Using a theorem prover can reduce the number of manipulation errors, but, given current technology, the amount of labor is still daunting. This research explores an alternative solution to the refinement problem, namely the use of syntactic transformations to realize each refinement step. We reduce formal labor by employing automatic transformations that guarantee the preservation of desirable properties — e.g., deadlock-freedom, safety, liveness. Automatic transformations are particularly appealing for the development of large, complex distributed systems, where a manual approach to refinement would be prohibitively expensive. Distributed computations are, by nature, reactive and concurrent, so their correctness cannot be specified as a simple functional relationship between inputs and outputs. Instead, specifications must describe the time-varying behavior of the system. Further difficulty is caused by the fact that such important characteristics of distribution as deadlock-freedom are global properties that cannot be achieved through considering local structures only. Transformations generally must encompass the entire system. This paper presents three syntactic transformations that can be used to replace an *atomic action* in a concurrent program by a *program fragment.* The work presented here is an extension and continuation of the transformations work presented in [Attie et. al. 96]. We give the *applicability conditions* for our transformations, and show that *deadlock-freedom* and certain *liveness* properties are preserved when the transformations are applied in a context where the applicability conditions are satisfied.

1 Introduction

Formal program verification is widely accepted as a means of guaranteeing the correctness of concurrent programs [Hoare 69, Francez 92, Lamport 80, Vardi 87]. The practical utility of formal verification is limited by numerous factors — for example, the large amount of manual labor required, the possibility of proof errors, the lack of personnel trained in proof techniques, and so on. It is also clear that post-development verification alone does not provide a systematic software development process. Successive refinement is an alternative approach for producing correct concurrent programs: start with an abstract specification and incrementally refine it to a stage where implementation becomes relatively straightforward. Refinement is not a new idea, of course, but most of the techniques proposed to date (for example, see [Back et. al. 83, Back et. al. 85, Chandy et. al. 88, Ramesh et. al. 87]) require large amounts of manual formal labor for each refinement step. Even methodologies based on automatic theorem proving [Manna et. al. 94, Constable et. al. 89, Cleaveland et. al. 96] require user intervention, either to select the rule of inference used to generate the next step of a proof [Cleaveland et. al. 96], or to supply invariants and/or correct automatically generated invariants that are not "inductive," i.e., that cannot be proven to be invariants in the deductive system being used [Manna et. al. 94]. Other approaches [Aceto 92, van Glabeek 90, Czaja et. al. 91] address the issue of which equivalence relations are preserved by refinement. In other words, if P and Q are programs such that P is "bisimilar" to Q (under some notion of bisimulation, see [Baeten et. al. 90, Milner 89]), then $ref(P)$ will be bisimilar to $ref(Q)$ under this same bisimulation notion, where $ref(P)$ and $ref(Q)$ are "corresponding" refinements of P, Q, i.e., refinements that refine the same action of P, Q in the same way. While such approaches provide a nice theoretical foundation, they do not directly address the central concern, namely the establishment of a relationship between a program and its refinement, i.e., between P and $ref(P)$.

Central to our approach is the concept of *correctness-preserving syntactic transformations*. Such transformations are mechanizable and, therefore, do not involve significant amounts of manual labor. Using this approach, the process of development may be viewed as the human-assisted high-level compilation of a specification into code. Furthermore, by avoiding proof-based methods, we obviate the need to formulate (usually) complicated invariants, a difficult task at best, even with the aid of automated tools.

In the foreseeable future, human creativity will remain essential for choosing an appropriate transformation to apply at each stage. But verifying that a transformation preserves desired properties is unnecessary, in our approach, because this is guaranteed by the fact that the transformations are correctness-preserving.

This paper presents transformations that decompose an action into a sequences and/or choices (possibly nested) of "smaller" actions. These transformations are *sound*, in that they preserve certain correctness properties of concurrent programs (i.e., if the initial program has the property, then so will the transformed program). The correctness properties that are preserved are deadlock-freedom and temporal leads-to (action a leads-to action b iff whenever a is executed, b is guaranteed to be subsequently executed). Formal proofs of soundness are given, as well as an example of refinement using the transformations.

2 Notation, Syntax, and Semantics

A *program* is the composition of a fixed set of sequential processes executing concurrently. We use the nondeterministic interleaving model of concurrency. That is, we view concurrency as the nondeterministic interleaving of events. An *event* is the atomic (i.e., indivisible) execution of an *action*. We use ;, $[]$, $||$ to denote sequence, choice, and parallel composition, respectively. The semantics of these operators is similar to that given in CSP [Hoare 85]. To model state transitions, we employ the concept of a labeled transition system, as used in [Milner 89]. $\xrightarrow{a}$ will denote the transition relation induced by action, a. The formal meaning of ;, $[]$, $||$, $\xrightarrow{a}$ is given below.

Definition 1 (*Action*)
An action, *a consists of a character string, (i.e., an identifier) drawn from some set, $\mathcal{A}$, of identifiers.*

We use lower-case letters towards the beginning of the alphabet to denote actions.

Definition 2 (*Action Expression*)
An action expression *E is a finite expression given by the following BNF grammar:*

<*action_expression*> ::=
<*action_expression*> $[]$ <*action_expression*> |
<*action_expression*> ; <*action_expression*> |
(<*action_expression*>) |
<*action*> | ε | 0

E, F, G, H range over the set of action expressions. We make the convention that ; has higher binding power than $[]$, so that $E; F \,[]\, G$ denotes $(E; F) \,[]\, G$. Intuitively, $E; F$ means execute E and then execute F, while $E \,[]\, F$ means execute either E or F. $[]$ is commutative, and $[]$, ; are both associative.

0, ("Stop"), is the identity element of $[]$, and ε ("Skip"), is the identity element of ;. They obey the following axioms: $0 \,[]\, A = A$, $A \,[]\, 0 = A$, $A; \varepsilon = A$, $\varepsilon; A = A$. We define the relation of equality (=) among action expressions as follows. $E = F$ iff one can be obtained from the other by a finite number of any of the following: 1) application of the above axioms for 0 and ε, 2) application of the commutativity property of $[]$ or the associativity property of $[]$ and ;, and 3) adding/removing parentheses in accordance with the precedence of ; over $[]$.

We define αE, the *alphabet* of action expression E, as follows.

Definition 3 (*Alphabet*)
The alphabet *of an action expression is given as follows*:

$alphabet(a) \stackrel{\text{df}}{=} \{a\}$
$alphabet(E \,[]\, F) \stackrel{\text{df}}{=} alphabet(E) \cup alphabet(F)$
$alphabet(E; F) \stackrel{\text{df}}{=} alphabet(E) \cup alphabet(F)$

Definition 4 (*Sequential Process*)
A sequential process *P_i consists of a process body and a process alphabet. The body of process P_i, $body(P_i)$, is an expression of the form $F_i; *E_i$ where F_i, E_i are action expressions. The alphabet of process P_i, $alphabet(P_i)$, is defined to be a set of action names. The process alphabet must contain $alphabet(F_i) \cup alphabet(E_i)$.*

Note that this definition extends the definition of "alphabet" to processes. We have also introduced "*", which denotes infinite iteration. We extend $=$ to process bodies in a straightforward manner. If $body(P_i) = F_i; *E_i$, and $body(P_j) = F_j; *E_j$, then $body(P_i) = body(P_j)$ iff $F_i = F_j$ and $E_i = E_j$. Finally, $P_i = P_j$ iff $alphabet(P_i) = alphabet(P_j)$ and $body(P_i) = body(P_j)$. (Note that when we write $alphabet(P_i) = alphabet(P_j)$, the $=$ symbol denotes standard set-theoretic equality, because alphabets are sets.)

Definition 5 (*Program*)
A program P *is the parallel composition of one or more sequential processes; i.e.,* $P \equiv (\| \; i \in \varphi : P_i)$, *where* φ *is some suitable index set. Also,* $alphabet(P) = (\cup i \in \varphi : alphabet(P_i))$.

$\|$ is commutative and associative, which justifies the index notation $\| \; i \in \varphi$ introduced in the above definition. For sake of simplicity, we assume that all variables in a program are uniquely named. We extend $=$ to programs in the expected manner: $(\| \; i \in \varphi : P_i) = (\| \; i \in \psi : Q_i)$ iff $\varphi = \psi$ and, for all $i \in \varphi, P_i = Q_i$.

Definition 6 (*Participant Set* PA_P)
The participant set $PA_P(a)$ *of action* a *is given by*:

$$PA_P(a) \stackrel{\text{df}}{=} \{i \mid a \in alphabet(P_i)\}$$

$PA_P(a)$ is the set of processes within program P that jointly and synchronously participate in the execution of action a. If $|PA_P(a)| > 1$ then a is a *multiparty interaction* of program P. If $|PA_P(a)| = 1$, then a is a *local action* of some process P_i (namely the P_i such that $a \in alphabet(P_i)$) in program P.

2.1 Operational Semantics

The operational semantics of a program P is defined by giving the transitions that the execution of each action a in $alphabet(P)$ can generate. Our definition proceeds bottom up, defining the binary transition relation $\stackrel{a}{\rightarrow}$ over action expressions first, then over sequential processes, and finally over programs. In each case, execution of a takes the action expression (sequential process, program) to a new action expression (sequential process, program resp.). In order to avoid the well-known phenomenon that the behavior of $E \parallel F$ and $\varepsilon; E \parallel \varepsilon; F$ is different even though they are "equal", we stipulate that the transition relation cannot be applied to 0 and ε, i.e., $\stackrel{\varepsilon}{\rightarrow}$ and $\stackrel{0}{\rightarrow}$ are not defined. This does not cause any difficulties, since ε and 0 can always be eliminated from an expression using the above axioms, after which the transition relation can be applied. This stipulation means that 0 and ε are never executed.

Definition 7 (*Transition Relation* $\stackrel{a}{\rightarrow}$)
The transitions generated by action a *are as follows*:

Act. $\dfrac{}{(a; E) \stackrel{a}{\rightarrow} E}$

Ch. $\dfrac{E \stackrel{a}{\rightarrow} E'}{(E \parallel F) \stackrel{a}{\rightarrow} E'} \qquad \dfrac{F \stackrel{a}{\rightarrow} F'}{(E \parallel F) \stackrel{a}{\rightarrow} F'}$

Seq $\dfrac{E \xrightarrow{a} E'}{(E;F) \xrightarrow{a} (E';F)}$

Iter $\dfrac{((E);*E) \xrightarrow{a} E'}{*E \xrightarrow{a} E'}$

We extend $\xrightarrow{a}$ *to processes by stipulating that* $P_i \xrightarrow{a} P_i'$ *iff* $body(P_i) \xrightarrow{a} body(P_i')$ *and* $alphabet(P_i) = alphabet(P_i')$. *In other words, the alphabets are the same and the bodies are related by* $\xrightarrow{a}$. *Finally, we extend* $\xrightarrow{a}$ *to programs as follows:*

Let $P \equiv (\| \; i \in \varphi : P_i)$, $P' \equiv (\| \; i \in \varphi : P_i')$. *Then* $P \xrightarrow{a} P'$ *iff:*

1. *for all* $i \in PA_P(a) : P_i \xrightarrow{a} P_i'$
2. *for all* $i \in \varphi - PA_P(a) : P_i \equiv P_i'$

Definition 8 (*Ready, Enabled, Disabled*)
For a process P_i, *we write* $P_i \xrightarrow{a}$ *to mean that there exists a* P_i' *such that* $P_i \xrightarrow{a} P_i'$. *We say that* P_i readies a *in this case.*

For a program P, *we write* $P \xrightarrow{a}$ *to mean that there exists a* P' *such that* $P \xrightarrow{a} P'$. *In this case, we say that* P enables a, *or that* a *is* enabled *in* P. *We also write* $P \overset{a}{\not\rightarrow}$ *to mean that there does not exist a* P' *such that* $P \xrightarrow{a} P'$, *and we say that* a *is* disabled *in* P *in this case.*

Suppose $P_i \xrightarrow{c}$. Then the general form for the body of P_i is $F;*E$, where F has one of the forms c, $c;G$, $c \parallel H$, $c;G \parallel H$. All of these forms are subsumed by the form $c;G \parallel H$ however, since $c = c;\varepsilon \parallel 0$, $c;G = c;G \parallel 0$, $c \parallel G = c;\varepsilon \parallel G$. Thus the introduction of 0 and ε allows us to avoid a large amount of tedious case-analysis. We now present some preliminary definitions and results.

Definition 9 (*Derivative, Path*)
If $P \xrightarrow{a_1} \cdots \xrightarrow{a_n} P'$ *for some sequence* $a_1, \ldots, a_n$ *of actions, then (again following [Milner 89]) we say that* P' *is a* derivative *of* P. *The sequence* $a_1, \ldots, a_n$ *is called a* path. *If path* $\pi = a_1, \ldots, a_n$, *then we abbreviate* $P \xrightarrow{a_1} \cdots \xrightarrow{a_n} P'$ *by* $P \xrightarrow{\pi} P'$.

A path is also called a *computation.*

Definition 10 (*Maximal Path*)
A path that is either infinite or ends in a derivative that has no enabled actions is called maximal.

Consider a program consisting of a single process $P_1 = *[a;b \parallel a;c]$. Clearly, $P_1 \xrightarrow{a} (b;P_1)$, and $P_1 \xrightarrow{a} (c;P_1)$. This example can easily be extended to arbitrary paths. Thus, establishing $P \xrightarrow{\pi} P'$ and $P \xrightarrow{\pi} P''$ for some P, π, P', P'', does not allow us to conclude $P' = P''$. Thus, if P and π are given, then the assertion $P \xrightarrow{\pi} P'$ can be regarded as an abbreviation for "let $P \xrightarrow{\pi} P'$ for some P'."

Suppose we have a path $\pi = \pi' bc \pi''$. Then, the path $\pi' cb \pi''$ is said to be obtained from π by a single *exchange* of actions b and c.

Definition 11 (*Independent*)
Two actions b, c are independent *in program P iff $PA_P(b) \cap PA_P(c) = \emptyset$*

Definition 12 (*Equivalent*)
Two paths π, ρ are equivalent *($\pi \equiv \rho$) iff one can be obtained from the other by a finite or countably infinite number of exchanges of adjacent independent actions (with the restriction that each action can be subjected to only a finite number of exchanges).*

Proposition 1 *If actions b, c are independent in program P, and $P \xrightarrow{bc} P'$, then $P \xrightarrow{cb} P'$.*

Proposition 2 *Let $P \xrightarrow{\pi} Q$. If π and ρ are equivalent, then $P \xrightarrow{\rho} Q$.*

3 Correctness Properties of Programs

As stated in the introduction, the correctness properties that our transformations preserve are deadlock-freedom and temporal leads-to. We define these properties as follows.

Definition 13 (*Deadlock-Freedom*)
If for every derivative P' of P, there is some action a such that $P' \xrightarrow{a}$, then P is deadlock-free.

As our concern here is with nonterminating, reactive, concurrent programs, the property of deadlock-freedom is a crucial one, and indeed is a prerequisite for demonstrating that our transformations preserve the temporal leads-to property.

Definition 14 (*Temporal Leads-to, $a \leadsto b$, $\models a \leadsto b$*)
A computation π satisfies $a \leadsto b$ iff every occurrence of a along π is eventually followed by an occurrence of b.

A program P satisfies $a \leadsto b$ iff every maximal computation of P satisfies $a \leadsto b$.

We write $\pi \models a \leadsto b, P \models a \leadsto b$ for π satisfies $a \leadsto b$, P satisfies $a \leadsto b$ respectively.

Temporal leads-to is a particular form of *liveness* property that is very useful in verifying that distributed systems interact properly with their environment. For example, every *request* must "lead-to" a suitable *response*. Temporal leads-to properties are also interesting because they can be easily composed. For example, if $a \leadsto b$, and $b \leadsto c$, then $a \leadsto c$ (i.e., $\leadsto$ is transitive). Thus, a leads-to property $a_1 \leadsto a_n$ can be established as a sequence of leads-to properties $a_1 \leadsto a_2$, $a_2 \leadsto a_3$, ..., $a_{n-1} \leadsto a_n$. Each of these intermediate leads-to properties would presumably be established by a single transformation, and then $a_1 \leadsto a_n$ would be established by the sequence of these transformations.

In establishing liveness properties, the notion of *fairness* is useful. A *fair scheduling notion* usually specifies that if an action is enabled "sufficiently often," then it is eventually executed (where different fairness notions have different specifications for "sufficiently often"). For our purposes, the following notion suffices.

Definition 15 (*Weak Action Fairness*)
The fairness notion weak action fairness, *denoted Φ, is as follows:*
if an action is enabled continuously from some point onwards, then it is eventually executed.

In the sequel, we shall use "fairness" as an abbreviation for weak action fairness.

Definition 16 (*Fair Computation*)
A computation π is fair *iff π has no suffix along which some action is enabled continuously but never executed.*

Definition 17 (*Fair $a \rightsquigarrow b$, $\models_\Phi a \rightsquigarrow b$*)
A program P satisfies $a \rightsquigarrow b$ with respect to weak fairness iff every maximal fair computation of P satisfies $a \rightsquigarrow b$.
We write $P \models_\Phi a \rightsquigarrow b$ for P satisfies $a \rightsquigarrow b$ with respect to weak fairness.

4 The Transformations

As stated in the introduction, our transformations decompose an action c into possibly nested sequences and/or choices of "smaller" actions. Thus, every occurrence of c in some process P_i is replaced by an action expression E_i that specifies the decomposition of P_i's part in action c. Since c has, in general, more than one participant process, we are lead to the following definition.

Definition 18 (*Program Fragment, Process Fragment*)
Let P be a program and $c \in alphabet(P)$. For each $i \in PA_P(c)$, let E_i be an action expression such that $alphabet(E_i) \cap alphabet(P) = \emptyset$. Then $E = (\parallel\ i \in PA_P(c) : E_i)$ is a program fragment for P with respect to c, *and each E_i is a* process fragment for P_i with respect to c.

We have identified three transformations that can be used to refine programs for concurrent systems (we take our "high-level" programs to be, in effect, executable specifications). Given a program fragment $E = (\parallel\ i \in \psi : E_i)$ for P with respect to c, our transformations are as follows:

1. The transformation $[c/c;E]$: every occurrence of c in P_i (for all $i \in PA_P(c)$) is replaced by $c; E_i$.
2. The transformation $[c/E;c]$: every occurrence of c in P_i (for all $i \in PA_P(c)$) is replaced by $E_i; c$.
3. The transformation $[c/E]$: every occurrence of c in P_i (for all $i \in PA_P(c)$) is replaced by E_i.

To facilitate the formal definition of these transformation, we first define our notion of *syntactic substitution.*

Definition 19 (*Syntactic Substitution*)
Let a be an arbitrary action, and E, G, H be arbitrary action expressions. Then, we have

$$\varepsilon[c/E] = \varepsilon$$
$$0[c/E] = 0$$
$$a[c/E] = a \text{ if } a \neq c$$
$$c[c/E] = E$$
$$(G \parallel H)[c/E] = ((G[c/E]) \parallel (H[c/E]))$$
$$(G;H)[c/E] = ((G[c/E]);(H[c/E]))$$

In the sequel, we will use the abbreviation G_t for $G[c/E]$ for an arbitrary action expression G.

The following definitions will also be useful in the subsequent technical discussion.

Definition 20 ($initial(E_i)$, $initial(E)$)
Let $E = (\| \ i \in \psi : E_i)$ *be a program fragment for* P *with respect to* c. *Then* $initial(E_i) \stackrel{\text{df}}{=\!=} \{a \mid E_i \stackrel{a}{\rightarrow}\}$, *and* $initial(E) \stackrel{\text{df}}{=\!=} \{a \mid E \stackrel{a}{\rightarrow}\}$.

In other words, $initial(E_i)$, $initial(E)$ are the sets of actions that are the first actions executable by E_i, E respectively.

We say that a process P_i *enters* E iff P_i executes an action in $initial(E)$.

Example 1 *If* $E = (E_1 :: a; b) \parallel (E_2 :: (b; d) \ [\!] \ a)$, *then* $initial(E_2) = \{a, b\}$, *and* $initial(E) = \{a\}$.

Definition 21 ($choice(P_i, c)$)
Let P_i *be a process and* $c \in alphabet(P_i)$. *Then* $choice(P_i, c) = \{d \mid$ "$c \ [\!] \ d$" occurs in $body(P_i)\}$.

In other words, $choice(P_i, c)$ is the set of all actions that P_i could execute as an alternative choice to executing c.

4.1 The Transformation $[c/c; E]$

Definition 22 (*Transformation* $[c/c; E]$)
We define the transformation $[c/c; E]$ *in a bottom-up manner as follows. For an arbitrary process* P_i *such that* $c \in alphabet(P_i)$, *and* $body(P_i) = H; *G$ *for some action expressions* H, G, *define* $P_i[c/c; E] = Q_i$, *where* $alphabet(Q_i) = alphabet(P_i) \cup alphabet(E)$, $body(Q_i) = H[c/c; E]; *(G[c/c; E])$.

Let $P = (\| \ i \in \varphi : P_i)$ *be an arbitrary program. We define*

$$P[c/c; E] = (\| \ i \in PA_P(c) : P_i[c/c; E]) \ \| \ (\| \ i \in \varphi - PA_P(c) : P_i).$$

4.2 The Transformation $[c/E; c]$

Definition 23 (*Transformation* $[c/E; c]$)
We define transformation $[c/E; c]$ *in a bottom-up manner as follows. For an arbitrary process* P_i *such that* $c \in alphabet(P_i)$, *and* $body(P_i) = H; *G$ *for some action expressions* H, G, *define* $P_i[c/E; c] = Q_i$, *where* $alphabet(Q_i) = alphabet(P_i) \cup alphabet(E)$, $body(Q_i) = H[c/E; c]; *(G[c/E; c])$.

Let $P = (\| \ i \in \varphi : P_i)$ *be an arbitrary program. We define*

$$P[c/E; c] = (\| \ i \in PA_P(c) : P_i[c/E; c]) \ \| \ (\| \ i \in \varphi - PA_P(c) : P_i).$$

4.3 The Transformation $[c/E]$

Definition 24 (*Transformation* $[c/E]$)
We define the transformation $[c/E]$ *in a bottom-up manner as follows. For an arbitrary process* P_i *such that* $c \in alphabet(P_i)$, *and* $body(P_i) = H; *G$ *for some action expressions* H, G, *define* $P_i[c/E] = Q_i$, *where* $alphabet(Q_i) = alphabet(P_i) \cup alphabet(E)$, $body(Q_i) = H[c/E]; *(G[c/E])$.

Let $P = (\| \ i \in \varphi : P_i)$ *be an arbitrary program. We define*

$$P[c/E] = (\| \ i \in PA_P(c) : P_i[c/E]) \ \| \ (\| \ i \in \varphi - PA_P(c) : P_i).$$

5 Soundness of the Transformations

All of our transformations have associated *applicability conditions* that determine when the transformations can be used. These applicability conditions are needed in order to avoid the following problems that can arise when applying a transformation:

- *Introduction of deadlock*: The original program is deadlock-free but the transformed program is deadlock-prone.
- *Partial execution*: The action c in the original program is *atomic*, i.e., either it is executed to completion or not at all. When c is refined into (for example) E, it is possible for situations to arise where E is only partially executed. This is undesirable, since an execution of c in the original program corresponds to a complete execution of E in the transformed program. A partial execution of E therefore, corresponds to no behavior of the original program. Hence the transformed program exhibits behavior that could never be exhibited by the original program. This makes it impossible (usually) to verify that the desired correctness properties embodied in the original program have been preserved by the transformed program.
- *Uncoordinated entry*: It is possible that some participants of c start executing E (in the refined program) but others never enter their corresponding parts of E.

In effect, these applicability conditions test for certain properties of programs that guarantee the absence of the problems discussed above.

5.1 The Applicability Conditions

We now present and define formally the *applicability conditions* that must be satisfied in order for our transformations to preserve the correctness properties of deadlock-freedom and temporal leads-to. We shall need the following definition.

Definition 25 ($\hat{E}$)
Let $E = (\parallel\ i \in \varphi : E_i)$ *be a program fragment. Then* $\hat{E} = (\parallel\ i \in \varphi : E_i; 0)$.

$\hat{E}$ is a program whose computations are those that can be generated by executing each process fragment E_i exactly once.

The *single iteration property* states that any partial execution of E (in isolation) can always be completed into a full execution of E. In other words, if E is regarded as a program and executed in isolation, then every process (i.e., the $E_i, i \in \varphi$) is guaranteed to execute its body to completion. This condition is crucial in establishing that neither *partial execution* nor *deadlock* (see above) occur in the transformed program.

Definition 26 (*The Single-Iteration Property*)
Let $E = (\parallel\ i \in \varphi : E_i)$ *be a program fragment. Then* E *has the single-iteration property iff for every derivative* F *of* $\hat{E}$, *there exists a path* π *such that* $F \xrightarrow{\pi} (\parallel\ i \in \varphi : 0)$.

Definition 27 (*The Loose-Synchronization Property*)
Let $E = (\parallel\ i \in \varphi : E_i)$ *be a program fragment. Then* E *has the loose-synchronization property iff* $\hat{E}$ *has no derivative of the form* $(\parallel\ i \in \varphi' : E_i; 0)\ \parallel\ (\parallel\ i \in \varphi'' : F_i; 0)\ \parallel\ (\parallel\ i \in \varphi''' : 0)$, *where* φ', φ''' *are nonempty, and* φ', φ'', φ''' *partition* φ.

In effect, the loose synchronization property states that it is impossible for a subset of the processes in φ to execute a complete iteration of their process fragments E_i while another subset has not yet started. Hence, all the participants of c are loosely synchronized in that at some point, they must simultaneously all be in E.

Definition 28 (*The No-Overtaking Property*)
Let $E = (\parallel i \in \varphi : E_i)$ be a program fragment. Then E has the no-overtaking property iff for every derivative of $\hat{E}$ of the form $(\parallel i \in \varphi' : F_i; 0) \parallel (\parallel i \in \varphi'' : 0)$, (where φ', φ'' partition φ), we have $(\cup_{i \in \varphi'} alphabet(F_i)) \cap (\cup_{j \in \varphi''} alphabet(E_j)) = \emptyset$.

In effect, the no-overtaking property states that it is impossible for a subset of the processes in φ to execute a complete iteration of their process fragments E_i and then loop around and interact with the other processes that have yet to complete the first iteration. This allows us to establish a "separation" between successive iterations of E: a process executing the n'th iteration of E cannot interact with a process executing the $n+1'$st iteration of E.

Our next applicability condition is *conspiracy-resistance.* The *conspiracy-resistance property* states that, if the processes that ready a particular action a are frozen (i.e., not allowed to execute any action), then that freezing does not prevent yet another participant of a from eventually readying a. This property is used in proving that partial execution does not occur in the transformed program (since all of the participants of E eventually ready E and so E is executed to completion.

For a more extensive discussion of conspiracy-resistance, the reader is referred to [Attie et. al. 93]. To formally define conspiracy resistance, we must first define the concept of (a, A)-derived *program.*

Definition 29 ((a, A)-derived *program*)
Let P be a program and let $a \in alphabet(P)$, and $A \subseteq PA_P(a)$. The (a, A)-derived *program $P_{a,A}$ is obtained from P by replacing with 0 every occurrence of b (where $b \neq a$) that occurs in a choice with a, in every process $P_i \in A$.*

Definition 30 (*Conspiracy Resistance*)
An action a is conspiracy resistant *in a program P iff for every computation π of P the following condition holds:*

> *Let π_1 be any finite prefix of π ending in the derivative P', and let $PA_a^{\mathcal{I}}$ be the set of all the participants of a that ready a in P'. Then, for every computation π_2 of the $(a, PA_a^{\mathcal{I}})$-derived program $P'_{a,PA_a^{\mathcal{I}}}$ obtained from P', there exists a participant $P_j \in (PA_a - PA_a^{\mathcal{I}})$ such that P_j eventually readies a along π_2.*

The third applicability condition is *coordinated-entry.* Conspiracy resistance guarantees that every participant of E eventually readies E. However, once E is readied, it is still possible for the participant process to execute actions not in E (if E occurs in a choice). The coordinated-entry condition guarantees that, if one of the participants of E has actually chosen E for execution, then they will all do so.

Definition 31 (*Coordinated Entry*)
We say that a program fragment $E = (\parallel i \in \varphi : E_i)$ is coordinated-entry *with respect to a program $P = (\parallel i \in \varphi : P_i)$ and an action $c \in alphabet(P)$ iff there exists a $\psi \subseteq \varphi$ such that:*

1. $\bigwedge i, j \in \psi : initial(E_i) = initial(E_j) \wedge initial(E_i) \subseteq initial(E)$
2. $\bigwedge i \in \varphi - \psi : initial(E_i) \cap initial(E) = \emptyset$
3. $\bigwedge i \in \varphi - \psi : choice(P_i, c) \cap (\cup_{i \in \psi} alphabet(P)) \neq \emptyset$

The first clause says that every initial action of E is an initial action of every E_i, $i \in \psi$. Hence, every process fragment E_i, $i \in \psi$, participates in every initial action of E. The second clause says that no initial action of E_i, $i \notin \psi$, is an initial action of E. In other words, the process fragments E_i, $i \notin \psi$ never participate in any initial action of E. These two clauses together imply that it is the process fragments in ψ that control the entry into E for all processes. The third clause says that every action that is a possible alternative choice to entering E must have some participant process with index in ψ. Thus, the alternatives to entering E are controlled by the processes in ψ. Hence, in the transformed program, if a process $P_i \in \psi$ arrives at the choice point where it can either enter E or execute an alternative action, there are two possibilities:

1. P_i enters E. In this case, every process in ψ must enter E simultaneously with P_i, by clause 1. Also, by clause 3, no process outside ψ can execute an action not in E upon reaching the choice point.
2. P_i executes an action not in E. In this case, no process in ψ can enter E upon reaching the choice point, since the processes in ψ must enter E together (by clause 1). By clause 2, no process outside ψ can enter E upon reaching the choice point, since these processes can only enter E by interacting with some process in ψ that has already entered E.

Hence we see that in both cases all processes make the same decision about whether or not to enter E.

We now give the exact applicability conditions for each transformation. We assume, in the rest of the paper (except the example), that these conditions are always met whenever the transformation is mentioned.

Definition 32 (*Applicability Conditions for the Transformation* $[c/c; E]$)
The applicability conditions for $[c/c; E]$ *are as follows:*

1. $alphabet(E) \cap alphabet(P) = \emptyset$.
2. *E has the single-iteration property.*

Definition 33 (*Applicability Conditions for the Transformation* $[c/E; c]$)
The applicability conditions for $[c/E; c]$ *are as follows:*

1. $alphabet(E) \cap alphabet(P) = \emptyset$.
2. *E has the single-iteration property.*
3. *c is conspiracy resistant in P.*
4. *E is coordinated-entry with respect to P and c.*

Definition 34 (*Applicability Conditions for the Transformation* $[c/E]$)
The applicability conditions for $[c/E]$ *are as follows:*

1. $alphabet(E) \cap alphabet(P) = \emptyset$.
2. *E has the single-iteration property.*
3. *c is conspiracy resistant in P.*
4. *E is coordinated-entry with respect to P and c.*
5. *E has the loose-synchronization property.*
6. *E has the no-overtaking property.*

5.2 Layering Results

Our proof strategy rests on the following lemmas, which show that for every computation of the transformed program, there exists an equivalent "layered" computation in which all the actions of (every iteration of) E are executed contiguously, i.e., with no actions outside E occurring in between two actions from E. This allows us to establish a natural correspondence between computations of the original program P and the transformed program Q : a computation π of P corresponds to every layered computation ρ of Q that results from π by replacing every execution of c in π by a contiguous execution of E. The correspondence is extended to unlayered computations of Q using the equivalence relation over computations: if π corresponds to ρ, and $\rho \equiv \rho'$, then π corresponds to ρ'. Since there is, in general, more than one way to execute E, the correspondence relation can be seen as relating a single computation of P to a countably infinite number of equivalence classes of computations of Q.

Lemma 3 *Let $Q = P[c/c; E]$. For every computation π of Q, there exists an equivalent layered computation π'.*

Lemma 4 *Let $Q = P[c/E; c]$. For every computation π of Q, there exists an equivalent layered computation π'.*

Lemma 5 *Let $Q = P[c/E]$. For every computation π of Q, there exists an equivalent layered computation π'.*

5.3 Deadlock-Freedom Results

Our deadlock freedom results are straightforward: all of our transformations preserve the property of deadlock freedom. Thus, if the original program is deadlock free, then so is the transformed program.

Theorem 6 *Let $Q = P[c/c; E]$. If P is deadlock-free, then so is Q.*

Theorem 7 *Let $Q = P[c/E; c]$. If P is deadlock-free, then so is Q.*

Theorem 8 *Let $Q = P[c/E]$. If P is deadlock-free, then so is Q.*

5.4 Temporal Leads-to Results

We assume, in this section, that P is deadlock free (and hence Q is too by the deadlock-freedom results above). We group the liveness results for each transformation. The first theorem of each group expresses the preservation (by the transformations) of leads-to properties satisfied by the original program. The second theorem of each group expresses the preservation (by the transformations) of leads-to properties satisfied by the program fragment E used in the transformation. For the third theorem of each group, we first need the following definition.

Definition 35 ($\Diamond$)
Let $E = (\parallel\ i \in \varphi : E_i)$ *be a program fragment, and let* $\hat{E} = (\parallel\ i \in \varphi : E_i; 0)$. *Then* $\hat{E} \models \Diamond b$ *iff every computation of* $\hat{E}$ *contains* b.

Now suppose the original program P satisfies $a \leadsto c$. This would then imply that in the transformed program Q, if a is executed, then eventually a derivative is reached where entering E is the only possible continuation (otherwise, there would be an alternative to c in P, and so $P \models a \leadsto c$ would not hold). Hence, if executing b is inevitable once E is entered, it then follows that a leads to b in Q. This is expressed by the third theorem of each group.

5.4.1 Liveness Results for The Transformation $[c/c; E]$

Theorem 9 *Let* $Q = P[c/c; E]$. *If* $PA_P(a) \cap PA_P(b) \neq \emptyset$ *and* $P \models_\Phi a \leadsto b$, *then* $Q \models_\Phi a \leadsto b$.

Theorem 10 *Let* $Q = P[c/c; E]$. *If* $PA_E(a) \cap PA_E(b) \neq \emptyset$ *and* $\hat{E} \models a \leadsto b$, *then* $Q \models_\Phi a \leadsto b$.

Theorem 11 *Let* $Q = P[c/c; E]$. *If* $P \models_\Phi a \leadsto c$, *and* $\hat{E} \models \Diamond b$, *then* $Q \models_\Phi a \leadsto b$.

5.4.2 Liveness Results for The Transformation $[c/E; c]$

Theorem 12 *Let* $Q = P[c/E; c]$. *If* $PA_P(a) \cap PA_P(b) \neq \emptyset$ *and* $P \models_\Phi a \leadsto b$, *then* $Q \models_\Phi a \leadsto b$.

Theorem 13 *Let* $Q = P[c/E; c]$. *If* $PA_E(a) \cap PA_E(b) \neq \emptyset$ *and* $\hat{E} \models a \leadsto b$, *then* $Q \models_\Phi a \leadsto b$.

Theorem 14 *Let* $Q = P[c/E; c]$. *If* $P \models_\Phi a \leadsto c$, *and* $\hat{E} \models \Diamond b$, *then* $Q \models_\Phi a \leadsto b$.

5.4.3 Liveness Results for The Transformation $[c/E]$

Theorem 15 *Let* $Q = P[c/E]$. *If* $PA_P(a) \cap PA_P(b) \neq \emptyset$ *and* $P \models_\Phi a \leadsto b$, *then* $Q \models_\Phi a \leadsto b$.

Theorem 16 *Let* $Q = P[c/E]$. *If* $PA_E(a) \cap PA_E(b) \neq \emptyset$ *and* $\hat{E} \models a \leadsto b$, *then* $Q \models_\Phi a \leadsto b$.

Theorem 17 *Let* $Q = P[c/E]$. *If* $P \models_\Phi a \leadsto c$, *and* $\hat{E} \models \Diamond b$, *then* $Q \models_\Phi a \leadsto b$.

6 Example: Mobile Cellular Phone System

We now illustrate the use of the transformations to establish deadlock freedom and progress properties. The example we use is a solution to the mobile cellular telephone handoff problem, for which we first give an informal description.

6.1 Informal Problem Description

A mobile telephone system has a fixed number, N, of mobile telephones (henceforth called mobiles), and a fixed number, M, of message switching centers (henceforth called msc's). Normally, each mobile has a radio link with exactly one msc, which is called that mobile's manager, all calls to the mobile being routed by trunk lines to this msc, and then by radio to the mobile. The mobile, however, may move away from the msc so that eventually the signal quality between the mobile and the msc deteriorates to an unacceptable level. When this happens, management of the mobile must be transferred to another msc with which it has a better signal. This transfer operation is called a handoff .

Informally, the system operates as follows: Each msc repeatedly performs a signal-level check on all mobiles that it handles. When a signal-level check indicates that the signal quality has deteriorated to an unacceptable level, the following events occur in sequence:

1. the msc synchronizes with all other msc's
2. all of the msc's perform a signal-level check with the mobile
3. an election is performed to determine the msc with the highest signal level
4. a handoff is performed between the old and new msc's

The interactions corresponding to these events are:

- *chk*: a managing msc interacts with a mobile to determine the strength of the signal between them. This is called a "signal-level check."
- *synch*: used to synchronize all of the msc's as a preliminary to electing a new msc to handle a particular mobile
- *psc*: a signal-level check performed prior to an election
- *el*: the election of a new msc
- *st*: the preliminary setting up of trunk lines just prior to a handoff
- *ho*: the handoff

There are no safety specifications in this system. The liveness specification may be stated informally as:

If the signal between a particular mobile and it's msc deteriorates to an unacceptable level, then, provided the mobile has not moved outside the area of coverage, it will eventually be handed off to an msc with whom it has adequate signal strength.

The problem description was obtained from the Electronic Industries Association Interim Standard, "Cellular Radiotelecommunication Intersystem Operations: Intersystem Handoff," [EIA87].

6.2 The Example

We consider a system consisting of one mobile (mb) and two msc's ($mc1, mc2$). Our initial high-level model of the system is given in figure 1. There are only two actions: $coord_1$ models the case where the mobile is being managed by $mc1$, and $coord_2$ models the case where the mobile is being managed by $mc2$.

We apply the transformation $[c/E]$ to program 1, where $c = coord_1$, and $E = (E_1 :: chk_1; (at_1 \parallel (bt_1; psc_1))) \parallel (E_2 :: chk_1; (at_1 \parallel (bt_1; synch_1; psc_1))) \parallel (E_3 :: at_1 \parallel (synch_1; psc_1))$.

We can easily check that the applicability conditions for $[c/E]$ are met. The resulting program 2 is shown in figure 2.

Next we apply the transformation $[c/c; E]$ to program 2, where $c = psc_1$, and $E = (E_1 :: ho_{11} \parallel ho_{12}) \parallel (E_2 :: el_1; (ho_{11} \parallel (st_{12}; ho_{12}))) \parallel (E_3 :: el_1; (ho_{11} \parallel (st_{12}; ho_{12})))$.

We can verify that the applicability conditions for $[c/c; E]$ are met. The resulting program 3 is shown in figure 3.

To complete the derivation of the program, we apply symmetric transformations to $coord_2$. The first of these is $[c/E]$ where $c = coord_2$, and $E = (E_1 :: chk_2; (at_2 \parallel (bt_2; psc_2))) \parallel (E_2 :: at_2 \parallel (synch_2; psc_2)) \parallel (E_3 :: chk_2; (at_2 \parallel (bt_2; synch_2; psc_2)))$.

Applying this to program 3 results in program 4, given in figure 4.

Finally, we apply the transformation $[c/c; E]$ to program 4, where $c = psc_2$, and $E = (E_1 :: ho_{22} \parallel ho_{21}) \parallel (E_2 :: el_2; (ho_{22} \parallel (st_{21}; ho_{21}))) \parallel (E_3 :: el_2; (ho_{22} \parallel (st_{21}; ho_{21})))$. The resulting program 5, given in figure 5, is our final program.

6.3 Correctness Properties of the Final Program

Deadlock-freedom of program 1 is trivially verified by inspection. Hence, by our deadlock-freedom results, we conclude that program 5 is deadlock-free. By using our liveness results, we conclude that program 5 satisfies the following leads-to properties. We list the relevant transformation to the left of the properties (we used four transformations, so we refer to them in sequence as transformations 1 through 4):

Transformation 1: $bt_1 \leadsto_\Phi psc_1$
Transformation 2: $psc_1 \leadsto_\Phi el_1$
Transformation 3: $bt_2 \leadsto_\Phi psc_2$
Transformation 4: $psc_2 \leadsto_\Phi el_2$

These properties can be composed together, using the transitivity of $\leadsto$. This then allows us to conclude the liveness properties that result from composing transformations:

Transformation 1 followed by transformation 2: $bt_1 \leadsto_\Phi el_1$
Transformation 3 followed by transformation 4: $bt_2 \leadsto_\Phi el_2$

We remark that, given program 5 only, the task of establishing the above correctness properties would not be altogether trivial. A complicated invariant would have to be established to prove deadlock-freedom of program 5, and an argument based on decreasing bound functions or "helpful directions" would be used to show liveness. Such arguments can be formalized using deductive systems for temporal logic [Manna et. al. 94]. The proofs however, are usually somewhat involved.

Finally, we note that program 5 is slightly sub-optimal: mc1 participates in at_2. Since at_2

```
mb:: *[   coord1
       [] coord2
      ]
||
mc1:: *[   coord1
        [] coord2
       ]
||
mc2:: *[   coord1
        [] coord2
       ]
```

Figure 1: Program 1

```
mb:: *[   chk1 ; [  at1
                 [] bt1;psc1
                ]
       []
          coord2
      ]
||
mc1:: *[   chk1 ; [  at1
                 [] bt1 ; synch1 ; psc1
                ]
        []
          coord2
       ]
||
mc2:: *[   [  at1
            [] synch1 ; psc1
           ]
        []
          coord2
       ]
```

Figure 2: Program 2

```
mb:: *[   chk1 ; [  at1
                 [] bt1 ; psc1 ; [  ho11
                                 [] ho12
                                 ]
                 ]
       []
          coord2
      ]
||
mc1:: *[   chk1 ; [  at1
                 [] bt1 ; synch1 ; psc1 ; el1 ; [  ho11
                                                [] st12 ; ho12
                                                ]
                 ]
       []
          coord2
      ]
||
mc2:: *[   [  at1
           [] synch1 ; psc1 ; el1 ; [  ho11
                                    [] st12 ; ho12
                                    ]
           ]
       []
          coord2
      ]
```

Figure 3: Program 3

```
mb:: *[   chk1 ; [  at1
                 [] bt1 ; psc1 ; [  ho11
                                 [] ho12
                                 ]
                 ]
       []
          chk2 ; [  at2
                 [] bt2 ; psc2
                 ]
      ]
||
mc1:: *[   chk1 ; [  at1
                 [] bt1 ; synch1 ; psc1 ; el1 ; [  ho11
                                                 [] st12 ; ho12
                                                 ]
                 ]
        []
          [  at2
          [] synch2 ; psc2
          ]
       ]
||
mc2:: *[   [  at1
           [] synch1 ; psc1 ; el1 ; [  ho11
                                    [] st12 ; ho12
                                    ]
           ]
        []
          chk2 ; [  at2
                 [] bt2 ; synch2 ; psc2
                 ]
       ]
```

Figure 4: Program 4

```
mb:: *[   chk1 ; [  at1
                 [] bt1 ; psc1 ; [  ho11
                                 [] ho12
                                ]
                ]
       []
          chk2 ; [  at2
                 [] bt2 ; psc2 ; [  ho22
                                 [] ho21
                                ]
                ]
      ]
||
mc1:: *[   chk1 ; [  at1
                 [] bt1 ; synch1 ; psc1 ; el1 ; [  ho11
                                                [] st12 ; ho12
                                               ]
                ]
       []
          [  at2
           []
             synch2 ; psc2 ; el2 ; [  ho22
                                   [] st21 ; ho21
                                  ]
          ]
      ]
||
mc2:: *[   [  at1
           [] synch1 ; psc1 ; el1 ; [  ho11
                                    [] st12 ; ho12
                                   ]
          ]
       []
          chk2 ; [  at2
                 [] bt2 ; synch2 ; psc2 ; el2 ; [  ho22
                                                [] st21 ; ho21
                                               ]
                ]
      ]
```

Figure 5: Program 5

represents the positive result of a signal-level check between mc2 and mb only, there is no need for mc1 to participate in at_2. However, the applicability conditions of the transformation $[c/E]$ required this participation. Eliminating this phenomenon would require designing transformations that permit *participant elimination*, i.e., an action c may be refined into a fragment E whose execution does not always require the participation of all the processes that would participate in the execution of c (in the original program). Such transformations are a topic for future work.

7 Future Work

In future work, we intend to address the topic of how to verify that the applicability conditions hold. Some of the conditions (coordinated-entry, $alphabet(E) \cap alphabet(P) = \emptyset$) are purely syntactic, and so can be checked algorithmically in an efficient manner. The remaining conditions (single-iteration, loose-synchronization, no-overtaking, conspiracy-resistance) are semantic. Checking them mechanically may incur exponential overhead. We plan to investigate alternative strategies. It may be possible to "construct" the fragment E, using a set of "derivation rules", so that E has the single-iteration and no-overtaking properties. Furthermore, one might then be able to show that, when E is constructed in this certain manner, that the property of conspiracy resistance is also preserved by the transformations. Thus, we are preserving a property (conspiracy-resistance) not because it is inherently an interesting program correctness property, but because it is an applicability condition for our transformations. In the proposal for this contract, we envisioned doing this when the applicability condition was a syntactic property. It has turned out to be useful to have more complex semantic properties as the applicability conditions for our transformations.

Devising a methodology for ensuring that the applicability conditions are met will allow us to reason much more powerfully about the results of applying sequences of transformations. Currently, we can infer the results of applying a sequence of transformations, but the intermediate steps of applying the transformations incur a manual verification of the applicability conditions. When these conditions are not met, we currently have little insight into why this is so, and how we can modify the derivation sequence of transformations to ensure that the applicability conditions for the next transformation are met.

8 Conclusions

In this paper we have described three transformations for program refinement. They are used for refining actions into nested sequences and choices of more refined actions, and may be viewed as tools for decomposing a large action into a sequence of smaller actions. Such decomposition is a natural step in the process of refining programs. We proved formally that our transformations preserve deadlock-freedom and temporal leads-to.

We note that the formal correctness proofs for the transformations are somewhat lengthy. The salient point, however, is that the proofs are, in effect, reused each time the transformations are applied. A more traditional proof rule for program correctness [Chandy et. al. 88, Francez 92, Lamport 80] has a shorter formal justification, but requires the designer to produce a manual proof each time the rule is applied. We believe that it is much more efficient to verify the correctness of a transformation that can be reused many times, even if the proof is somewhat lengthy.

Correctness-preserving transformations for distributed systems are, in principle, a foundation for the eventual goal of compiling abstract specifications into architecturally adequate code. Those

who find that objective too distant should, nevertheless, be interested in the medium-term goal of automating certain laborious and error-prone parts of the development process. An interactive compiler that handles much of the labor — and is guaranteed not to introduce the deadlocks and other errors that plague concurrent systems — would be valuable, even if it still depends heavily on human design creativity. This research is designed to support both the medium- and the long-term goals.

A Proofs of the Theorems

This section presents the proofs of all propositions, lemmas, and theorems in the paper. The following definitions will be useful:

An *event* is the execution of an action. The $n'th$ execution of an action a along some computation π will be denoted by a^n.

If a^n, b^m are two events along π, with a^n preceding b^m, then we denote the portion of π between a^n and b^m (including a^n, b^m) by $\pi(a^n, b^n)$.

Let Q_i be a process which contains the action expression E_i inits body. We say Q_i *is at* E_i is Q_i readies the actions in $initial(E_i)$. We say Q_i *is in* E_i if Q_i readies some action of E_i not in $initial(E_i)$.

Let $Q = P[c/c; E]$. If π is a layered computation of Q, then $\pi < c; E/c >$ denotes the computation that results from removing all events of E from π.

Let $Q = P[c/E; c]$. If π is a layered computation of Q, then $\pi < E; c/c >$ denotes the computation that results from removing all events of E from π.

Let $Q = P[c/E]$. If π is a layered computation of Q, then $\pi < E/c >$ denotes the computation that results from removing all events of E from π.

If Q results from P by applying any of the three transformations, then let E^n denote the $n'th$ iteration of E (along a particular computation π of Q). We say a^n, b^n are *adjacent* in π with respect to E^n if there is no event from E^n in the portion of π between a^n and b^n. (We assume, without loss of generality, that a^n precedes b^n along π.)

Proposition 1 *If actions b, c are independent in program P, and $P \xrightarrow{bc} P'$, then $P \xrightarrow{cb} P'$.*

Proof: Since b and c are independent in P, we have $PA_P(b) \cap PA_P(c) = \emptyset$ by definition 11. Let $P = (\| \; i \in \varphi : P_i)$, and P'' be such that $P \xrightarrow{b} P'' \xrightarrow{c} P'$. Then, by definition 7,

$$P'' = (\| \; i \in PA_P(b) : P_i'') \; \| \; (\| \; i \in \varphi - PA_P(b) : P_i), \quad \text{where } P_i \xrightarrow{b} P_i'' \text{ for all } i \in PA_P(b)$$

Hence, by definition 7 and $PA_P(b) \cap PA_P(c) = \emptyset$,

$$P' = (\| \; i \in PA_P(b) : P_i'') \; \| \; (\| \; i \in PA_P(c) : P_i') \; \| \; (\| \; i \in \varphi - (PA_P(b) \cup PA_P(c)) : P_i),$$
where $P_i \xrightarrow{c} P_i'$ for all $i \in PA_P(c)$

By $P_i \xrightarrow{c} P_i'$ for all $i \in PA_P(c)$ and definition 7,

$$P \xrightarrow{c} P''' \text{ where } P''' = (\| \; i \in PA_P(c) : P_i') \; \| \; (\| \; i \in \varphi - PA_P(c) : P_i)$$

Then, by $P_i \xrightarrow{b} P_i''$ for all $i \in PA_P(b)$ and definition 7,

$$P''' \xrightarrow{b} (\| \; i \in PA_P(c) : P_i') \; \| \; (\| \; i \in PA_P(b) : P_i'') \; \| \; (\| \; i \in \varphi - (PA_P(b) \cup PA_P(c)) : P_i)$$

Hence $P''' \xrightarrow{b} P'$, and so $P \xrightarrow{c} P''' \xrightarrow{b} P'$. Thus $P \xrightarrow{cb} P'$. □

Proposition 2 *Let $P \xrightarrow{\pi} Q$. If π and ρ are equivalent, then $P \xrightarrow{\rho} Q$.*

Proof: The proof is by induction on the number m of exchanges of independent adjacent actions required to obtain ρ from π.

Base Case: $m = 1$.

Now ρ is obtained from π by one exchange. Hence we can write $\pi = \pi' ab \pi''$, $\rho = \pi' ba \pi''$, where a, b are the exchanged independent actions. Thus we have

$$P \xrightarrow{\pi'} P' \xrightarrow{ab} P'' \xrightarrow{\pi''} Q \qquad (*)$$

for some P', P''.

Since a, b are independent in P, we can apply proposition 1 to $P' \xrightarrow{ab} P''$, thereby concluding $P' \xrightarrow{ba} P''$. Using this result and (*) we have $P \xrightarrow{\pi'} P' \xrightarrow{ba} P'' \xrightarrow{\pi''} Q$. Hence $P \xrightarrow{\rho} Q$. Thus the base case is established.

Induction Step: $m = n + 1$, $n \geq 1$, where the inductive hypothesis is assumed for n exchanges.

Since ρ is obtained from π by $n+1$ exchanges, there must exist a η such that η is obtained from π by n exchanges, and ρ is obtained from η by one exchange. By the inductive hypothesis, we have $P \xrightarrow{\eta} Q$. Since ρ is obtained from η by one exchange, we use same argument as employed in the base case (i.e., for a single exchange) to conclude $P \xrightarrow{\rho} Q$. This establishes the induction step. □

Lemma 3 *Let $Q = P[c/c; E]$. For every computation π of Q, there exists an equivalent layered computation π'.*

Proof: Let a^n, b^n denote the events in π corresponding to the execution of a, b in the n'th iteration of E. Assume that a^n, b^n are adjacent in π with respect to E^n, and, without loss of generality, assume that a^n precedes b^n along π. Let d^m be an arbitrary event in $\pi(a^n, b^n)$. By definition of *adjacent*, d^m is not an event of E^n. Because c synchronizes entry to E, d cannot be an event of $E^m, m \neq n$. Hence d^m is an event of P, i.e., $d \in alphabet(P)$. Since c^n occurs before a^n along π, we have the following ordering:

$c^n \; a^n \; d^m \; b^n$

Now any $P_i \in PA_Q(b)$ must also be in $PA_Q(c)$ by definition of $[c/c; E]$. Hence P_i cannot be a participant in d since $d \notin alphabet(E)$ (and so d^m cannot be an event of E^n, which it must be if P_i participates in it). Hence d and b are independent actions and so can be commuted. Since d^m is an arbitrary event between a^n and b^n, all such events can be commuted. Hence a^n, b^n can be made strictly adjacent, i.e., with no other events at all in between them. Since a^n, b^n is an arbitrary pair that is adjacent with respect to E^n, it follows that all such pairs can be brought together. Hence we can produce an equivalent computation in which the events of E^n form a single contiguous subsequence. Repeating this operation for all E^n (i.e., for all values of n) gives us the computation π'. □

Lemma 4 *Let $Q = P[c/E; c]$. For every computation π of Q, there exists an equivalent layered computation π'.*

Proof: Let a^n, b^n denote the events in π corresponding to the execution of a, b in the n'th iteration of E. Assume that a^n, b^n are adjacent in π with respect to E^n, and, without loss of generality, assume that a^n precedes b^n along π. Let d^m be an arbitrary event in $\pi(a^n, b^n)$. By definition of *adjacent*, d^m is not an event of E^n. Because c synchronizes exit from E, d cannot be an event of $E^m, m \neq n$. Hence d^m is an event of P, i.e., $d \in alphabet(P)$. Since c^n occurs before a^n along π, we have the following ordering:

$a^n \; d^m \; b^n \; c^n$

Now any $P_i \in PA_Q(b)$ must also be in $PA_Q(c)$ by definition of $[c/E; c]$. Hence P_i cannot be a participant in d since $d \notin alphabet(E)$ (and so d^m cannot be an event of E^n, which it must be if P_i participates in it). Hence d and a are independent actions and so can be commuted. Since d^m is

an arbitrary event between a^n and b^n, all such events can be commuted. Hence a^n, b^n can be made strictly adjacent, i.e., with no other events at all in between them. Since a^n, b^n is an arbitrary pair that is adjacent with respect to E^n, it follows that all such pairs can be brought together. Hence we can produce an equivalent computation in which the events of E^n form a single contiguous subsequence. Repeating this operation for all E^n (i.e., for all values of n) gives us the computation π'. □

Lemma 5 *Let $Q = P[c/E]$. For every computation π of Q, there exists an equivalent layered computation π'.*

Proof: Let a^n, b^n denote the events in π corresponding to the execution of a, b in the n'th iteration of E. Assume that a^n, b^n are adjacent in π with respect to E^n, and, without loss of generality, assume that a^n precedes b^n along π. Let d^m be an arbitrary event in $\pi(a^n, b^n)$. By definition of *adjacent*, d^m is not an event of E^n. By the no-overtaking condition, d^m cannot be an event of $E^m, m \neq n$. Hence d^m is an event of P, i.e., $d \in alphabet(P)$. Hence, we have the following ordering:

$a^n \; d^m \; b^n$

Now any $P_i \in PA_Q(d)$ cannot be a participant in both a and b since $d \notin alphabet(E)$ (and so d^m cannot be an event of E^n, which it must be if P_i participates in a, b, and d). Furthermore, if a and d have some process in common, then d^m follows a^n causally along π. If follows that no participant P_i of d can execute b^n after it has executed d^m, since this would involve P_i's leaving E^n to execute d^m and then re-entering E^n to execute b^n. By construction of $[c/E]$, this behavior is not possible. Hence, we conclude that either a and d are independent, or b and d are independent. Hence, d^m can be commuted with either a^n or b^n. Since d^m is an arbitrary event between a^n and b^n, all such events can be commuted in this way, and so a^n, b^n can be made strictly adjacent, i.e., with no other events at all in between them. Since a^n, b^n is an arbitrary pair that is adjacent with respect to E^n, it follows that all such pairs can be brought together. Hence we can produce an equivalent computation in which the events of E^n form a single contiguous subsequence. Repeating this operation for all E^n (i.e., for all values of n) gives us the computation π'. □

Theorem 6 *Let $Q = P[c/c; E]$. If P is deadlock-free, then so is Q.*

Proof: Let Q' be an arbitrary derivative of Q, i.e., $Q \xrightarrow{\pi} Q'$ for some computation π. By lemma 3, there exists a layered computation π' of Q such that $\pi' \equiv \pi$. Hence, by proposition 2, $Q \xrightarrow{\pi'} Q'$. There are two cases.

Case 1: no Q'_i in Q' is at E_i or in E_i. By a projection argument, we can show that the computation $\rho = \pi' < c; E/c >$ is a computation of P. Let $P \xrightarrow{\rho} P'$. Since P is deadlock free, $P' \xrightarrow{a}$ for some action a. We can also show that P' and Q' have the same continuations. Hence $Q' \xrightarrow{a}$.
(end of case 1)

Case 2: Some Q'_i in Q' is at E_i or in E_i.
Hence π' can be written as $\rho c \rho'$ (where ρ, ρ' could be empty). Also, $Q' = (\| \; i \in \psi : F_i; Q'_i) \; \| \; (\| \; i \in \psi' : Q'_i)$ where ψ contains all the processes at E_i or in E_i and ψ' contains the remaining processes. By $alphabet(E) \cap alphabet(P) = \emptyset$ and the presence of action c, which synchronizes entry to E_i, we can show that $F = (\| \; i \in \psi : F_i; 0)$ is a derivative of $\hat{E}$. Hence, by the single-iteration property of E, $F \xrightarrow{a}$ for some action a. Hence $Q' \xrightarrow{a}$.
(end of case 2)

Since $Q' \xrightarrow{a}$ in both cases, and Q' is an arbitrary derivative of Q, we conclude that Q is deadlock-

free. □

Theorem 7 *Let $Q = P[c/E; c]$. If P is deadlock-free, then so is Q.*

Proof: Let Q' be an arbitrary derivative of Q, i.e., $Q \xrightarrow{\pi} Q'$ for some computation π. By lemma 3, there exists a layered computation π' of Q such that $\pi' \equiv \pi$. Hence, by proposition 2, $Q \xrightarrow{\pi'} Q'$. There are two cases.

Case 1: no Q'_i in Q' is at E_i or in E_i. By a projection argument, we can show that the computation $\rho = \pi' < c; E/c >$ is a computation of P. Let $P \xrightarrow{\rho} P'$. Since P is deadlock free, $P' \xrightarrow{a}$ for some action a. We can also show that P' and Q' have the same continuations. Hence $Q' \xrightarrow{a}$.
(end of case 1)

Case 2: Some Q'_i in Q' is at E_i or in E_i.
Hence π' can be written as $\rho c \rho'$ (where ρ, ρ' could be empty). Also, $Q' = (\| \ i \in \psi : F_i; Q'_i) \ \| \ (\| \ i \in \psi' : Q'_i)$ where ψ contains all the processes at E_i or in E_i and ψ' contains the remaining processes. By the conspiracy-resistance, and coordinated-entry conditions, and the exit-synchronization provided by action c, we are guaranteed that eventually, every participant of c will enter E. Hence, any computation π'' of Q' eventually reaches a derivative Q'' of the form $Q' = (\| \ i \in PA_c(Q) : F_i; Q''_i) \ \| \ (\| \ i \in \varphi - PA_c(Q) : Q''_i)$.

By $alphabet(E) \cap alphabet(P) = \emptyset$, we can show that $F = (\| \ i \in PA_Q(c) : F_i; 0)$ is a derivative of $\hat{E}$. Hence, by the single-iteration property of E, $F \xrightarrow{a}$ for some action a. Hence $Q'' \xrightarrow{a}$.
(end of case 2)

Since Q' is an arbitrary derivative of Q, and $Q' \xrightarrow{a}$ in the first case, and Q' is guaranteed to always generate a derivative of the form of Q'' in the second case, we conclude that it is impossible for a derivative of Q that has no enabled actions to be generated. Hence q is deadlock-free. □

Theorem 8 *Let $Q = P[c/E]$. If P is deadlock-free, then so is Q.*

Proof: Let Q' be an arbitrary derivative of Q, i.e., $Q \xrightarrow{\pi} Q'$ for some computation π. By lemma 4, there exists a layered computation π' of Q such that $\pi' \equiv \pi$. Hence, by proposition 2, $Q \xrightarrow{\pi'} Q'$. There are two cases.

Case 1: no Q'_i in Q' is at E_i or in E_i. By a projection argument, we can show that the computation $\rho = \pi' < c; E/c >$ is a computation of P. Let $P \xrightarrow{\rho} P'$. Since P is deadlock free, $P' \xrightarrow{a}$ for some action a. We can also show that P' and Q' have the same continuations. Hence $Q' \xrightarrow{a}$.
(end of case 1)

Case 2: Some Q'_i in Q' is at E_i or in E_i.
Hence π' can be written as $\rho c \rho'$ (where ρ, ρ' could be empty). Also, $Q' = (\| \ i \in \psi : F_i; Q'_i) \ \| \ (\| \ i \in \psi' : Q'_i)$ where ψ contains all the processes at E_i or in E_i and ψ' contains the remaining processes. By the conspiracy resistance, coordinated-entry, and no-overtaking conditions, we are guaranteed that eventually, every participant of c will enter E. Hence, any computation π'' of Q' eventually reaches a derivative Q'' of the form $Q' = (\| \ i \in PA_c(Q) : F_i; Q''_i) \ \| \ (\| \ i \in \varphi - PA_c(Q) : Q''_i)$.

By $alphabet(E) \cap alphabet(P) = \emptyset$, we can show that $F = (\| \ i \in PA_Q(c) : F_i; 0)$ is a derivative of $\hat{E}$. Hence, by the single-iteration property of E, $F \xrightarrow{a}$ for some action a. Hence $Q'' \xrightarrow{a}$.
(end of case 2)

Since Q' is an arbitrary derivative of Q, and $Q' \xrightarrow{a}$ in the first case, and Q' is guaranteed to always generate a derivative of the form of Q'' in the second case, we conclude that it is impossible

for a derivative of Q that has no enabled actions to be generated. Hence q is deadlock-free. □

Theorem 9 *Let $Q = P[c/c; E]$. If $PA_P(a) \cap PA_P(b) \neq \emptyset$ and $P \models_\Phi a \rightsquigarrow b$, then $Q \models_\Phi a \rightsquigarrow b$.*

Proof: Let $Q \xrightarrow{\pi} Q' \xrightarrow{a^n\pi'}$, where $\pi a^n \pi'$ is an arbitrary maximal fair computation of Q with some occurrence of a along it. By lemma 3, there exists a layered computation ρ equivalent to $\pi a^n \pi'$. Hence, by a projection argument, $\rho < c; E/c >$ is a fair computation of P. Since $P \models_\Phi a \rightsquigarrow b$, we conclude that $\rho < c; E/c >$ contains an event b^m following a^n. Since a, b are not independent, π' also contains b^m following a^n (since a, b cannot be commuted). Since $\pi a^n \pi'$ was chosen arbitrarily, it follows that $Q \models_\Phi a \rightsquigarrow b$. □

Theorem 10 *Let $Q = P[c/c; E]$. If $PA_E(a) \cap PA_E(b) \neq \emptyset$ and $\hat{E} \models a \rightsquigarrow b$, then $Q \models_\Phi a \rightsquigarrow b$.*

Proof: Let $Q \xrightarrow{\pi} Q' \xrightarrow{a^n\pi'}$, where $\pi a^n \pi'$ is an arbitrary maximal fair computation of Q with some occurrence of a along it. By lemma 3, there exists a layered computation ρ equivalent to $\pi a^n \pi'$.

By fairness, $\hat{E} \models a \rightsquigarrow b$, and the entry-synchronization provided by c, we can show that every layered computation satisfies $a \rightsquigarrow b$. Hence $\rho \models a \rightsquigarrow b$. Hence ρ contains b^n following a^n. Since a, b are not independent (and therefore cannot be commuted), π' must contain b^n. Since $\pi a^n \pi'$ was chosen arbitrarily, it follows that $Q \models_\Phi a \rightsquigarrow b$. □

Theorem 11 *Let $Q = P[c/c; E]$. If $P \models_\Phi a \rightsquigarrow c$, and $\hat{E} \models \Diamond b$, then $Q \models_\Phi a \rightsquigarrow b$.*

Proof: Let $Q \xrightarrow{\pi}$, where π is an arbitrary maximal fair computation of Q. By lemma 3, there exists a layered computation ρ equivalent to $\pi a^n \pi'$. By a projection argument, $\rho < c; E/c >$ is a fair computation of P. Since $P \models_\Phi a \rightsquigarrow b$, we conclude that $\rho < c; E/c > \models_\Phi a \rightsquigarrow c$. Hence, by a projection argument, we conclude that every occurrence of a along π is followed eventually by entry into E. By fairness and $\hat{E} \models \Diamond b$, this in turn leads to execution of b, Hence $\pi \models a \rightsquigarrow b$. Since π was chosen arbitrarily, it follows that $Q \models_\Phi a \rightsquigarrow b$. □

Theorem 12 *Let $Q = P[c/E; c]$. If $PA_P(a) \cap PA_P(b) \neq \emptyset$ and $P \models_\Phi a \rightsquigarrow b$, then $Q \models_\Phi a \rightsquigarrow b$.*

Proof: Let $Q \xrightarrow{\pi} Q' \xrightarrow{a^n\pi'}$, where $\pi a^n \pi'$ is an arbitrary maximal fair computation of Q with some occurrence of a along it. By lemma 4, there exists a layered computation ρ equivalent to $\pi a^n \pi'$. Hence, by a projection argument, $\rho < E; c/c >$ is a fair computation of P. Since $P \models_\Phi a \rightsquigarrow b$, we conclude that $\rho < E; c/c >$ contains an event b^m following a^n. Since a, b are not independent, π' also contains b^m following a^n (since a, b cannot be commuted). Since $\pi a^n \pi'$ was chosen arbitrarily, it follows that $Q \models_\Phi a \rightsquigarrow b$. □

Theorem 13 *Let $Q = P[c/E; c]$. If $PA_E(a) \cap PA_E(b) \neq \emptyset$ and $\hat{E} \models a \rightsquigarrow b$, then $Q \models_\Phi a \rightsquigarrow b$.*

Proof: Let $Q \xrightarrow{\pi} Q' \xrightarrow{a^n\pi'}$, where $\pi a^n \pi'$ is an arbitrary maximal fair computation of Q with some occurrence of a along it. By lemma 4, there exists a layered computation ρ equivalent to $\pi a^n \pi'$.

By fairness, $\hat{E} \models a \rightsquigarrow b$, and the entry-synchronization provided by the conspiracy-resistance and coordinated-entry conditions, we can show that every layered computation satisfies $a \rightsquigarrow b$. Hence $\rho \models a \rightsquigarrow b$. Hence ρ contains b^n following a^n. Since a, b are not independent (and therefore cannot be commuted), π' must contain b^n. Since $\pi a^n \pi'$ was chosen arbitrarily, it follows that

$Q \models_\Phi a \rightsquigarrow b$. □

Theorem 14 *Let* $Q = P[c/E; c]$. *If* $P \models_\Phi a \rightsquigarrow c$, *and* $\hat{E} \models \Diamond b$, *then* $Q \models_\Phi a \rightsquigarrow b$.

Proof: Let $Q \xrightarrow{\pi}$, where π is an arbitrary maximal fair computation of Q. By lemma 4, there exists a layered computation ρ equivalent to $\pi a^n \pi'$. By a projection argument, $\rho < c; E/c >$ is a fair computation of P. Since $P \models_\Phi a \rightsquigarrow b$, we conclude that $\rho < c; E/c > \models_\Phi a \rightsquigarrow c$. Hence, by a projection argument, and the entry-synchronization provided by the conspiracy-resistance and coordinated-entry conditions, we conclude that every occurrence of a along π is followed eventually by entry into E. By fairness and $\hat{E} \models \Diamond b$, this in turn leads to execution of b, Hence $\pi \models a \rightsquigarrow b$. Since π was chosen arbitrarily, it follows that $Q \models_\Phi a \rightsquigarrow b$. □

Theorem 15 *Let* $Q = P[c/E]$. *If* $PA_P(a) \cap PA_P(b) \neq \emptyset$ *and* $P \models_\Phi a \rightsquigarrow b$, *then* $Q \models_\Phi a \rightsquigarrow b$.

Proof: Since the proof of theorem 12 above did not use the fact that action c provides exit-synchronization to E, then the same proof carries over here. □

Theorem 16 *Let* $Q = P[c/E]$. *If* $PA_E(a) \cap PA_E(b) \neq \emptyset$ *and* $\hat{E} \models a \rightsquigarrow b$, *then* $Q \models_\Phi a \rightsquigarrow b$.

Proof: Since the proof of theorem 13 above did not use the fact that action c provides exit-synchronization to E, then the same proof carries over here. □

Theorem 17 *Let* $Q = P[c/E]$. *If* $P \models_\Phi a \rightsquigarrow c$, *and* $\hat{E} \models \Diamond b$, *then* $Q \models_\Phi a \rightsquigarrow b$.

Proof: Since the proof of theorem 14 above did not use the fact that action c provides exit-synchronization to E, then the same proof carries over here. □

References

[Aceto 92] ACETO, L., 1992, Action Refinement in Process Algebras. D. Phil. thesis, University of Sussex, (Cambridge University Press).

[Attie et. al. 93] ATTIE, P., FRANCEZ, N., and GRUMBERG, O., Fairness and Hyperfairness in Multiparty Interactions, *Distributed Computing*, vol. 6, July 1993, pp. 245 – 254. Extended abstract appears in *Proceedings of the Seventeenth Annual Association of Computing Machinery Symposium on the Principles of Programming Languages*, San Francisco, January 1990, pp. 292 – 305.

[Attie et. al. 96] ATTIE, P., and DAS, C., Automating the Refinement of Specifications via Syntactic Transformations, to appear, *International Journal of Systems Science*, special issue on distributed systems.

[Back et. al. 83] BACK, R.J.R., and KURKI-SUONIO, R., 1989, Decentralization of Process Nets With Centralized Control. *Distributed Computing* 3: 73–87, (1989).

[Back et. al. 85] BACK, R.J.R., and KURKI-SUONIO, R., 1985, Serializability in Distributed Systems With Handshaking. Carnegie Mellon University TR 85-109.

[Baeten et. al. 90] BAETEN, J. C., and WEIJLAND, P., 1990, Process Algebra. Cambridge Tracts in Theoretical Computer Science.

[Chandy et. al. 88] CHANDY, K.M., and MISRA, J., 1988, Parallel Program Design. Addison-Wesley.

[Cleaveland et. al. 96] CLEAVELAND, R., and PANANGADEN, P., Type Theory and Concurrency. *International Journal of Parallel Programming* 17(2): 153 – 206, (1988).

[Constable et. al. 89] CONSTABLE, R., and HOWE, D., Nuprl as a General Logic. *Technical Report 89-1021, Department of Computer Science, Cornell University.*

[Czaja et. al. 91] CZAJA, I., VAN GLABEEK, R., and GOLTZ, U., 1991, Interleaving Semantics and Action Refinement with Atomic Choice. Arbeitspapiere der GMD 594, Sankt Augustin.

[Davis 87] DAVIS, N., 1987, Proceedings of the Fourth International Workshop on Software Specification and Design. IEEE Computer Society Press.

[EIA87] Electronic Industries Association Interim Standard, "Cellular Radiotelecommunications Intersystem Operations: Intersystem Handoff," document EIA/IS-41.2, November, 1987.

[Francez 92] FRANCEZ, N., 1992, Program Verification. Addison-Wesley.

[van Glabeek 90] VAN GLABEEK, R., 1990, Comparative Concurrency Semantics and Refinement of Actions. Ph.D. dissertation, Vrije Univerity of Amsterdam.

[Hoare 69] HOARE, C.A.R. Hoare, 1969, An Axiomatic Basis for Computer Programming. *CACM* 12 (10): 576–580.

[Hoare 78] HOARE, C.A.R., 1978, Communicating Sequential Processes. *CACM* 21 (8): 666–678.

[Hoare 85] HOARE, C.A.R., 1985, Communicating Sequential Processes. Prentice-Hall.

[Lamport 80] LAMPORT, L., 1980, The 'Hoare Logic' of Concurrent Programs. *Acta Informatica*, 14: 21–37.

[Manna et. al. 94] MANNA, Z., ANUCHITANUKUL, A., BJORNER, N., BROWNE, A., CHANGG, E., COLON, M., de ALFARO, L., DEVARJAN, H., SIPMA, H., and UNRIBE, T, STeP: the Stanford temporal prover. *Technical Report, Dept. of Computer Science, Stanford University.*

[Milner 89] MILNER, R., 1989, Communication and Concurrency. Prentice-Hall.

[Ramesh et. al. 87] RAMESH, S., and MEHNDIRATTA, H., 1987, A methodology for developing distributed programs. *IEEE Transactions on Software Engineering* SE-13 (8): 967–976.

[Vardi 87] VARDI, M.Y., 1987, Verification of Concurrent Programs, The Automata Theoretic Framework. *Logic in Computer Science.*

*U.S. GOVERNMENT PRINTING OFFICE: 1998-610-130-61221